L2Plan2: Learning Generalised Policies via Evolutionary Computation

Michelle Galea and **John Levine**
Department of Computer & Information Sciences
University of Strathclyde
Glasgow G1 1XH, UK

Abstract

L2Plan2 is an evolution-inspired system for inducing generalised planning policies from training data. A policy is here defined as a list of rules that specify which actions to be performed under which conditions. A policy is domain-specific and is used in conjunction with an inference mechanism that stipulates which rule to apply and which variable bindings to implement in order to formulate plans for problems within that domain.

Introduction

We present *L2Plan2*, an evolution-inspired system that induces generalised policies from available solutions to planning problems. The term generalised policy was coined by Martin & Geffner (2004) for a function that maps pairs of initial and goal states to actions. The actions outputted should, when performed, achieve the specified goal state from the specified initial state.

The name *L2Plan2* is in recognition of early work (Khardon 1999) in the induction of policies which has influenced the current representation used to describe learned knowledge (Khardon's system was called L2ACT). The approach to learning this knowledge is however directly inspired by the work of Koza (1992) and Spector (1994), in applying Genetic Programming to the evolution of lisp-like problems for solving various blocksworld problems.

The policies induced by *L2Plan2* encapsulate a specific type of control knowledge – they consist of a list of IF-THEN rules and each rule describes the conditions necessary for a particular domain operator/action template to be applied. Rules within a policy are ordered and the action of the first rule that may be applied is performed. If more than one valid combination of variable bindings exists then orderings on the variables and their values are adopted and the first valid combination is effected.

The next section describes how policies are learned, while the following describes how they are applied to solving planning problems.

The Learner

The induction of policies is carried out using Evolutionary Computation (EC) in a supervised learning context. EC is the application of methods inspired by Darwinian principles of evolution to computationally difficult problems, such as search and combinatorial optimisation. Evolutionary algorithms in general re-iteratively apply genetic-inspired operators to a population of solutions, with fitter individuals of a generation (according to some predefined fitness criteria) more likely to be selected for modification and insertion into successive generations than weaker members. On average, therefore, each new generation is expected to be fitter than the previous one.

Input to *L2Plan2* consists of an untyped STRIPS domain description and domain examples on which to evaluate the policies being learned. The output is a domain-specific policy that is used in conjunction with an inference mechanism to solve problems within that domain.

Figure 1 presents an outline of the system. Each iteration starts with an initial randomly-generated population of policies. The performance of these policies is evaluated on training data generated from planning problems from the domain under consideration. The resulting measure of fitness for a policy is used to determine whether it is replicated in the next iteration – a predefined number of policies with the highest fitness are inserted straightaway into the next generation. While the new generation remains underfilled, individuals from the current generation are selected and modified and inserted into the new generation. The system terminates if a predefined maximum number of generations have been created, or a policy attains maximum fitness by correctly solving all examples.

Since the results of the evaluation process influence the creation of the next generation, the average fitness of all policies is expected to improve from one generation to the next. The fact that several policies are in each iteration allows the possibility of exploring different regions of the solution space at once. This, coupled with an element of randomness that is used in the selection of policies for crossover and mutation, may help to prevent all policies from converging to a local optimum solution.

The following paragraphs describe in more detail the creation of the initial population, policy evaluation, and the genetic operators used to create new policies from old. Table 1 lists the settings of *L2Plan2* parameters.

Generating the Initial Population

L2Plan2 first generates an initial – the first generation – population of policies. The number of individuals in a population

```
(1) Create initial population
(2) WHILE termination criteria false
(3)   Evaluate current generation
(4)   Select n fittest individuals
(5)   Perform local search on selected individuals
(6)   Insert individuals into next generation
(7)   WHILE new generation not full
(8)     With probability P_c
(9)       Select 2 individuals (parents)
(10)      Perform crossover to produce 2 offspring
(11)      Perform mutation (with prob) on offspring
(12)      IF elitism=true
(13)        Select fittest 2 of parents and offspring
(14)      ELSE Select offspring
(15)      ENDIF
(16)    With probability 1 − P_c
(17)      Select 1 individual
(18)      Perform mutation on individual
(19)    Perform local search on selected individual/s
(20)    Insert individual/s into next generation
(21)  ENDWHILE
(22) ENDWHILE
(23) Output fittest individual
```

Figure 1: Pseudocode outline of *L2Plan*

Table 1: *L2Plan2* parameter settings

Parameter	Setting
Population size	100
Maximum number of generations	100
Number of fittest policies replicated	1
Crossover probability	0.9
Crossover elitism	true
Mutation probability	0.3
Local search branching	10
Local search depth	10
Tournament selection size	2
Minimum number of goal conditions	1
Maximum number of goal conditions	3

is predefined by the user and stays fixed until the system terminates. The number of rules in a policy is randomly set to be between one and twice the number of operators in the domain.

Each IF-THEN rule is also generated randomly. A rule takes the form:

`IF condition AND goalCondition THEN action`

with each of `condition` and `goalCondition` being a conjunction of unground literals, and where the literals in `condition` relate to the current state and those in `goalCondition` to the goal state. The action, i.e. the THEN part of the rule is first selected randomly from all domain actions.

The size of `goalCondition` is determined by drawing a random integer between user-defined minimum and maximum values, which determines the number of predicates chosen from the domain description. A predicate is first selected, and then the appropriate number of variables are randomly selected from all possible variables (no typing). Predicates are randomly negated.

The size of `condition` is currently determined by the number of parameters of the selected action, and a random selection of predicates. A predicate is selected randomly, and

```
(:rule position_briefcase_to_pickup_misplaced_object
  :condition (and (at ?x ?to))
  :goalCondition (and (not(at ?x ?to)))
  :action movebriefcase ?bc ?from ?to)
```

Figure 2: Example of a briefcase rule with a variable in `condition` that is not a parameter of the action

then variables for the predicate are randomly selected from the action's parameter list, with the addition of one extra variable, say called `?x`. Predicates are selected, and variables assigned, until all of an action's parameters and the additional variable are present in a predicate of `condition`. Each predicate is randomly negated. The extra variable increases the knowledge that can be expressed by a rule; without it, for instance, the learning process would be unable to discover rules such as the one shown in Fig. 2; this rule specifies that if an object is misplaced (i.e. its current location is not the location specified for it in the goal state), then a briefcase is moved to the current location of the object.

Note that a policy need not contain a rule to describe each action in the domain, and that the initially set number of rules for a policy, and the size of individual rules is liable to change with the application of genetic operators.

Evaluating a Policy

The training data on which a policy is evaluated is composed of a number of examples that are generated from a number of planning problems. Each example consists of a state encountered on a plan for the problem from which it is extracted, and a number of actions which may be taken from that state, each with an associated cost.

Consider a planning problem that includes an initial state S_I and a goal state S_G. A solution is found using an available planner; currently this is FF (Hoffmann & Nebel 2001) whose solutions are not guaranteed to be optimal. The length of the solution is determined and this is considered to be the benchmark length for this problem. Each possible action that may be taken from S_I is performed, leading to new states. For each new state a solution that attains S_G is found, again using FF. The length of each new solution is determined and compared to the benchmark length. The first example extracted from this problem is now created by attaching a cost to all possible actions from S_I, where the cost for each action is the difference between the length of the plan resulting from that action, and the original benchmark length. Figure 3 shows the representation used for a training example, which is consistent as far as possible with STRIPS syntax.

For each state resulting from each action of the original FF solution the same procedure is followed as for S_I, i.e. all possible actions from the next state on the original plan, say S_n, are performed, solutions from each of the resulting states to S_G are found, and costs for each possible action taken from S_n are determined from the solutions' lengths. Each training problem therefore yields as many examples as there are states encountered on the original FF plan.

The fitness of a policy is determined by averaging its performance over all examples, where for each example presented it is scored based on whether the selected action forms

```
(define (example blocks1_1)
(:domain blocksworld)
(:objects 5 4 3 2 1)
(:initial  ...  )
(:goal  ...  )
(:actions
  (move-b-to-b 1 3 4) 1
  (move-b-to-b 1 3 5) 1
  (move-b-to-b 4 2 1) 1
  (move-b-to-b 4 2 5) 1
  (move-b-to-t 1 3) 0
  (move-b-to-t 4 2) 0
  (move-t-to-b 5 1) 2
  (move-t-to-b 5 1) 2) )
```

Figure 3: A training example generated from a blocksworld problem

part of a known shorter or longer plan. Formula (1) below describes the fitness function where m is the number of training examples and $actionCost_i$ is the cost of the action taken by the policy for training example i:

$$fitness = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{1 + actionCost_i} \quad (1)$$

Creating a New Generation of Policies

Current *L2Plan2* settings are such that the fittest individual of the current generation is automatically replicated into the next generation, after a local search procedure has been performed (Fig. 1 lines 4–6).

The remainder of the next generation is populated by repeatedly selecting (with replacement) individuals from the current generation, modifying them and inserting the resulting individuals into the next generation (lines 7–21). With a certain probability crossover (also known as recombination) is performed (line 8), or reproduction (line 16). If the first case then two individuals (parents) are selected from the current population and a crossover operation is carried out producing two new individuals (offspring). With a predefined probability mutation is peformed on the offspring. If crossover elitism is set, then the fittest two individuals of the parents and offspring are selected, local search is performed on them, and then they are inserted into the next generation. Otherwise the offspring are selected and inserted into the next generation after local search has been performed.

If crossover is not carried out then a single individual from the current generation is selected for reproduction (copying). Mutation may be performed, then local search and the resulting individual is inserted into the next generation.

There are three types of crossover operations and six types of mutations. When crossover or mutation is to be performed the specific type applied is selected randomly from those described below:

Single Point Rule Level Crossover A crossover point is randomly chosen in each of the two policies, with valid points being before any of the rules (points need not be the same in the two policies). Two offspring policies are then created by merging part of the policy of one parent (as delineated by the crossover point), with a part of the other parent (the first part of parent A with the second part of parent B, and the second part of parent A with the first part of parent B).

Single Rule Swap Crossover A randomly selected rule from policy A is swapped with a randomly selected rule from policy B, resulting in two new policies. The replacing rule is inserted in the same position in the policy as the one it is replacing.

Similar Action Rule Crossover Two rules with the same action are randomly selected from two parent policies, one from each. Two new rules are created from the rules by randomly selecting a point in the IF- part of each rule and swapping over parts of the `condition` and/or `goalCondition` with each other. The new rules replace the originals in the parent policies.

Rule Addition Mutation A new rule is generated and inserted at a random position in the policy.

Rule Deletion Mutation A randomly selected rule is removed from the policy (if the policy contains more than one rule).

Rule Swap Mutation Two randomly selected rules have their positions swapped in a policy (if the policy has more than one rule).

Rule Literal Addition A predicate is randomly selected from the domain, populated with variables, and inserted into a randomly selected rule of a policy.

Rule Literal Deletion A randomly selected literal in a randomly selected rule of a policy is deleted.

Rule Condition Replacement The IF- part of a randomly selected rule of a policy is deleted and a new part is generated randomly to replace it.

Selection of individuals for crossover and reproduction is done using tournament selection with a size of two (Miller & Goldberg 1995) – this biases the selection towards individuals that have higher fittness values (exploitation of learned knowledge). However, randomness still plays a part in their selection and in the application of some genetic operators (those involving the selection of individual rules in a policy), in an attempt to avoid local minima (and encourage exploration of new areas of the solution space).

The local search procedure currently used is aimed at increasing the fitness of a policy as quickly as possible. It performs rule condition mutations – rule literal addition/deletion, rule condition replacement – on a policy a predefined number of times, called the local search branching factor. If no mutant is fitter than the original policy then the search stops. If a mutant is fitter then it replaces the original policy. The search is repeated (assuming a mutant is fitter than an original policy) a predefined maximum number of times, called the local search depth factor.

The Planner

In this work we abstract Martin's & Geffner's view of generalised policies and make a distinction between a policy – the knowledge used to solve a problem, and the inference mechanism that utilises the policy – the decision procedure that dictates when and how the knowledge is applied. Figure 4 presents a simplified view of a planner based on this distinction. A domain model defines a specific domain in terms of relevant objects, predicates, operators and their effects.

The policies induced by *L2Plan2* consist of a list of IF-THEN rules where each rule describes the conditions nec-

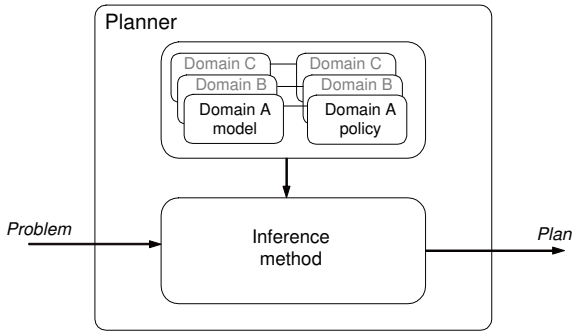

Figure 4: Planning using policies and inference mechanisms

essary for a particular domain operator to be applied. If variable-value bindings exist such that ground literals in `condition` match with the current state, and ground literals in `goalCondition` match with the goal state, then the action *may* be performed. Note though that the action's precondition must also be satisfied in the current state.

Which rule is actually applied and which variable-value bindings are implemented is decided by the inference mechanism. The list of rules is ordered and the first applicable rule is used. Variable and domain orderings are followed if more than one combination of bindings is valid.

Current induced policies may not encapsulate sufficient learned knowledge to solve all problems within a domain – before the goal is achieved they may be unable to suggest an action in a particular state, or degenerate into a looping behaviour executing the same two actions over and over. If such a situation is reached then a backup planner – FF – is used to suggest the next action. Planning then reverts back to the use of the *L2Plan2* policy.

References

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:263–302.

Khardon, R. 1999. Learning action strategies for planning domains. *Artificial Intelligence* 113:125–148.

Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Bradford Book, The MIT Press.

Martin, M., and Geffner, H. 2004. Learning generalized policies from planning examples using concept languages. *Applied Intelligence* 20:9–19.

Miller, B. L., and Goldberg, D. E. 1995. Genetic algorithms, tournament selection, and the effects of noise. Technical Report 95006, Department of General Engineering, University of Illinois at Urbana-Champaign, Urbana, IL.

Spector, L. 1994. Genetic programming and AI planning systems. In *Proc. 12th National Conference on Artificial Intelligence (AAAI-94)*, 1329–1334.